21 Calling remote APIs with IHttpClientFactory

This chapter covers

- Problems caused by using `HttpClient` incorrectly to call HTTP APIs
- Using `IHttpClientFactory` to manage `HttpClient` lifetimes
- Encapsulating configuration and handling transient errors with `IHttpClientFactory`

So far in this book we've focused on creating web pages and exposing APIs. Whether that's customers browsing a Razor Pages application or client-side SPAs and mobile apps consuming your APIs, we've been writing the APIs for others to consume.

However, it's very common for your application to interact with third-party services by consuming *their* APIs. For example, an e-commerce site needs to take payments, send email and SMS messages, and retrieve exchange rates from a third-party service. The most common approach for interacting with services is using HTTP. So far in this book we've looked at how you can *expose* HTTP services, using API controllers, but we haven't looked at how you can *consume* HTTP services.

In section 21.1 you'll learn the best way to interact with HTTP services using `HttpClient`. If you have any experience with C#, it's very likely you've used this class to send HTTP requests, but there are two gotchas to think about; otherwise your app could run into difficulties.

IHttpClientFactory was introduced in .NET Core 2.1; it makes creating and managing HttpClient instances easier and avoids the common pitfalls. In section 21.2 you'll learn how IHttpClientFactory achieves this by managing the HttpClient handler pipeline. You'll learn how to create *named clients* to centralize the configuration for calling remote APIs and how to use *typed clients* to encapsulate the remote service's behavior.

Network glitches are a fact of life when you're working with HTTP APIs, so it's important for you to handle them gracefully. In section 21.3 you'll learn how to use the open source resilience and fault-tolerance library Polly to handle common transient errors using simple retries, with the possibility for more complex policies.

Finally, in section 21.4 you'll see how you can create your own custom HttpMessageHandler handler managed by IHttpClientFactory. You can use custom handlers to implement cross-cutting concerns such as logging, metrics, or authentication, where a function needs to execute every time you call an HTTP API. You'll also see how to create a handler that automatically adds an API key to all outgoing requests to an API.

To misquote John Donne, "no app is an island," and the most common way of interacting with other apps and services is over HTTP. In .NET, that means using HttpClient.

21.1 Calling HTTP APIs: The problem with HttpClient

In this section you'll learn how to use HttpClient to call HTTP APIs. I'll focus on two common pitfalls in using HttpClient—socket exhaustion and DNS rotation problems—and show why they occur. In section 21.2 you'll see how to avoid these issues by using IHttpClientFactory.

It's very common for an application to interact with other services to fulfill its duty. Take a typical e-commerce store, for example. In even the most basic version of the application, you will likely need to send emails and take payments using credit cards or other services. You *could* try to build that functionality yourself, but it probably wouldn't be worth the effort.

Instead, it makes far more sense to delegate those responsibilities to third-party services that specialize in that functionality. Whichever service you use, they will almost certainly expose an HTTP API for interacting with the service. For many services, that will be the *only* way.

RESTful HTTP vs. gRPC vs. GraphQL

There are many ways to interact with third-party services, but HTTP RESTful services are still the king, decades after HTTP was first proposed. Every platform and programming language you can think of includes support for making HTTP requests and handling responses. That ubiquity makes it the go-to option for most services.

Despite their ubiquity, RESTful services are not perfect. They are relatively verbose, which means more data ends up being sent and received than with some other protocols. It can also be difficult to evolve RESTful APIs after you have deployed them. These limitations have spurred interest in two alternative protocols in particular: gRPC and GraphQL.

gRPC is intended to be an efficient mechanism for server-to-server communication. It builds on top of HTTP/2 but typically provides much higher performance than traditional RESTful APIs. gRPC support was added in .NET Core 3.0 and is receiving many performance and feature updates. For a comprehensive view of .NET support, see the documentation at https://docs.microsoft.com/aspnet/core/grpc.

While gRPC is primarily intended for server-to-server communication, GraphQL is best used to provide evolvable APIs to mobile and SPA apps. It has become very popular among frontend developers, as it can reduce the friction involved in deploying and using new APIs. For details, I recommend *GraphQL in Action* by Samer Buna (Manning, 2021).

Despite the benefits and improvements gRPC and GraphQL can bring, RESTful HTTP services are here to stay for the foreseeable future, so it's worth making sure you understand how to use them with `HttpClient`.

In .NET we use the `HttpClient` class for calling HTTP APIs. You can use it to make HTTP calls to APIs, providing all the headers and body to send in a request, and reading the response headers and data you get back. Unfortunately, it's hard to use correctly, and even when you do, it has limitations.

The source of the difficultly with `HttpClient` stems partly from the fact that it implements the `IDisposable` interface. In general, when you use a class that implements `IDisposable`, you should wrap the class with a `using` statement whenever you create a new instance. This ensures that unmanaged resources used by the type are cleaned up when the class is removed.

```
using (var myInstance = new MyDisposableClass())
{
    // use myInstance
}
```

That might lead you to think that the correct way to create an `HttpClient` is shown in the following listing. This shows a simple example where an API controller calls an external API to fetch the latest currency exchange rates and returns them as the response.

WARNING Do not use `HttpClient` as it's shown in listing 21.1. Using it this way could cause your application to become unstable, as you'll see shortly.

Listing 21.1 The incorrect way to use `HttpClient`

```
[ApiController]
public class ValuesController : ControllerBase
{
    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        using (HttpClient client = new HttpClient())
        {
            client.BaseAddress
                = new Uri("https://api.exchangeratesapi.io");
            var response = await client.GetAsync("latest");

            response.EnsureSuccessStatusCode();
            return await response.Content.ReadAsStringAsync();
        }
    }
}
```

Wrapping the HttpClient in a using statement means it is disposed of at the end of the using block.

Make a GET request to the exchange rates API.

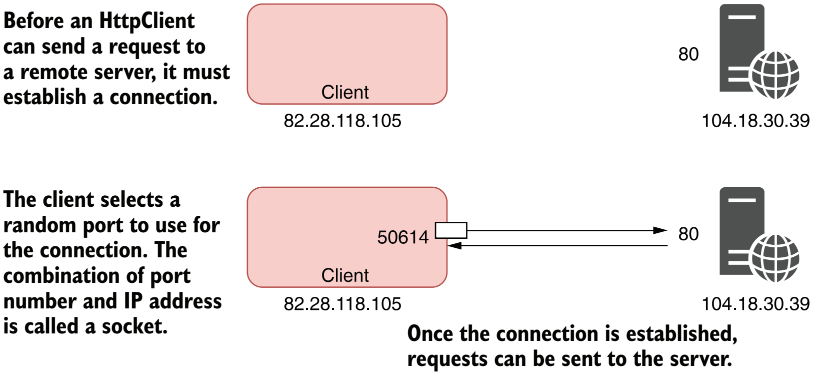
Configure the base URL used to make requests using the HttpClient.

Throw an exception if the request was not successful.

Read the result as a string and return it from the action method.

`HttpClient` is special, and you *shouldn't* use it like this! The problem is primarily due to the way the underlying protocol implementation works. Whenever your computer needs to send a request to an HTTP server, you must create a *connection* between your computer and the server. To create a connection, your computer opens a port, which has a random number between 0 and 65,535, and connects to the HTTP server's IP address and port, as shown in figure 21.1. Your computer can then send HTTP requests to the server.

DEFINITION The combination of IP address and port is called a *socket.*



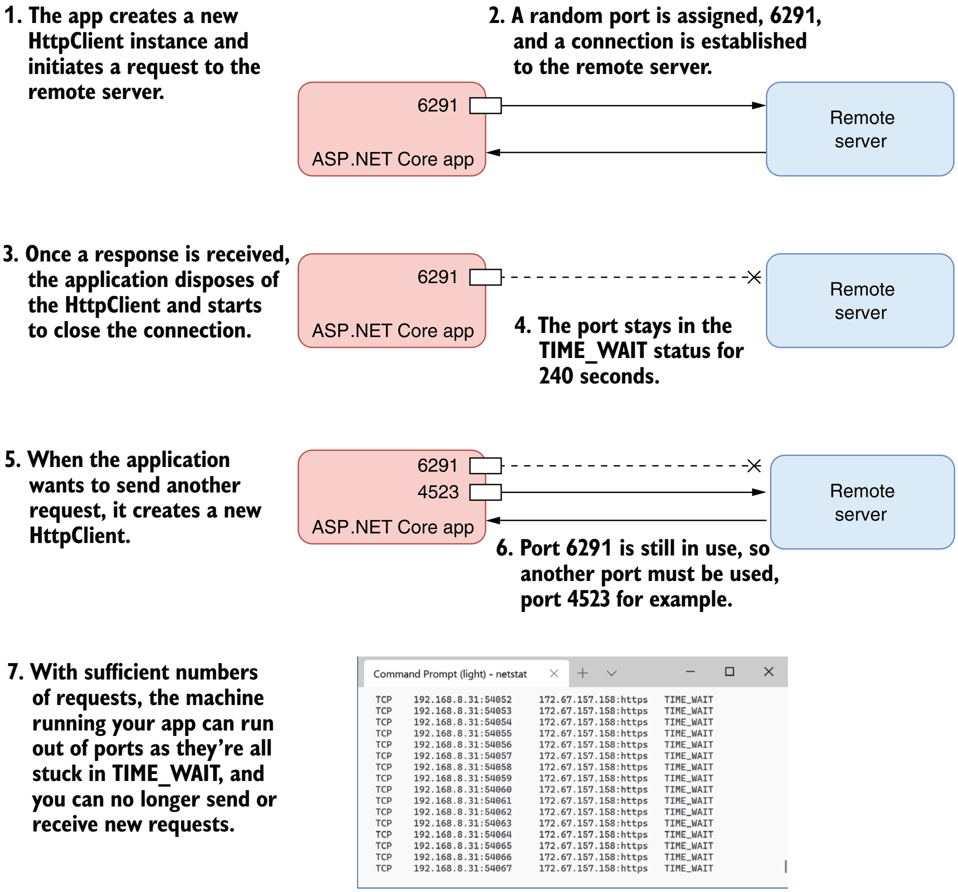
Before an HttpClient can send a request to a remote server, it must establish a connection.

The client selects a random port to use for the connection. The combination of port number and IP address is called a socket.

Once the connection is established, requests can be sent to the server.

Figure 21.1 To create a connection, a client selects a random port and connects to the HTTP server's port and IP address. The client can then send HTTP requests to the server.

The main problem with the `using` statement and `HttpClient` is that it can lead to a problem called *socket exhaustion*, as illustrated in figure 21.2. This happens when all the ports on your computer have been used up making other HTTP connections, so your computer can't make any more requests. At that point, your application will hang, waiting for a socket to become free. A very bad experience!


1. The app creates a new HttpClient instance and initiates a request to the remote server.
2. A random port is assigned, 6291, and a connection is established to the remote server.
6291
ASP.NET Core app
Remote server
3. Once a response is received, the application disposes of the HttpClient and starts to close the connection.
6291
ASP.NET Core app
Remote server
4. The port stays in the TIME_WAIT status for 240 seconds.
5. When the application wants to send another request, it creates a new HttpClient.
6291
4523
ASP.NET Core app
Remote server
6. Port 6291 is still in use, so another port must be used, port 4523 for example.
7. With sufficient numbers of requests, the machine running your app can run out of ports as they're all stuck in TIME_WAIT, and you can no longer send or receive new requests.
Command Prompt (light) - netstat

```
TCP    192.168.8.31:54052     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54053     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54054     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54055     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54056     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54057     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54058     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54059     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54060     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54061     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54062     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54063     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54064     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54065     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54066     172.67.157.158:https   TIME_WAIT
TCP    192.168.8.31:54067     172.67.157.158:https   TIME_WAIT
```

Figure 21.2 Disposing of `HttpClient` can lead to socket exhaustion. Each new connection requires the operating system to assign a new socket, and closing a socket doesn't make it available until the `TIME_WAIT` period of 240 seconds has elapsed. Eventually you can run out of sockets, at which point you can't make any outgoing HTTP requests.

Given that I said there are 65,536 different port numbers, you might think that's an unlikely situation. It's true, you will likely only run into this problem on a server that is making a lot of connections, but it's not as rare as you might think.

The problem is that when you dispose of an `HttpClient`, *it doesn't close the socket immediately*. The design of the TCP/IP protocol used for HTTP requests means that after trying to close a connection, the connection moves to a state called `TIME_WAIT`. The connection then waits for a specific period (240 seconds on Windows) before closing the socket completely.

Until the `TIME_WAIT` period has elapsed, you can't reuse the socket in another `HttpClient` to make HTTP requests. If you're making a lot of requests, that can quickly lead to socket exhaustion, as shown in figure 21.2.

> **TIP** You can view the state of active ports/sockets in Windows and Linux by running the command `netstat` from the command line or a terminal window. Be sure to run `netstat -n` on Windows to skip DNS resolution.

Instead of disposing of `HttpClient`, the general advice (before `IHttpClientFactory` was introduced in .NET Core 2.1) was to use a single instance of `HttpClient`, as shown in the following listing.

Listing 21.2 Using a singleton `HttpClient` to avoid socket exhaustion

```
[ApiController]
public class ValuesController : ControllerBase
{
    private static readonly HttpClient _client = new HttpClient
    {
        BaseAddress = new Uri("https://api.exchangeratesapi.io")
    };

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        var response = await _client.GetAsync("latest");

        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

A single instance of HttpClient is created and stored as a static field.

Multiple requests use the same instance of HttpClient.

This solves the problem of socket exhaustion. As you're not disposing of the `HttpClient`, the socket is not disposed of, so you can reuse the same port for multiple requests. No matter how many times you call `GetRates()` in the preceding example, you will only use a single socket. Problem solved!

Unfortunately, this introduces a different problem, primarily around DNS. DNS is how the friendly hostnames we use, such as manning.com, are converted into the IP addresses that computers need. When a new connection is required, the `HttpClient` first checks the DNS record for a host to find the IP address and then makes the connection. For subsequent requests, the connection is already established, so it doesn't make another DNS call.

For singleton `HttpClient` instances, this can be a problem because the `HttpClient` won't detect DNS changes. DNS is often used in cloud environments for load balancing to do graceful rollouts of deployments.[1] If the DNS record of a service you're calling changes during the lifetime of your application, a singleton `HttpClient` will keep calling the old service, as shown in figure 21.3.

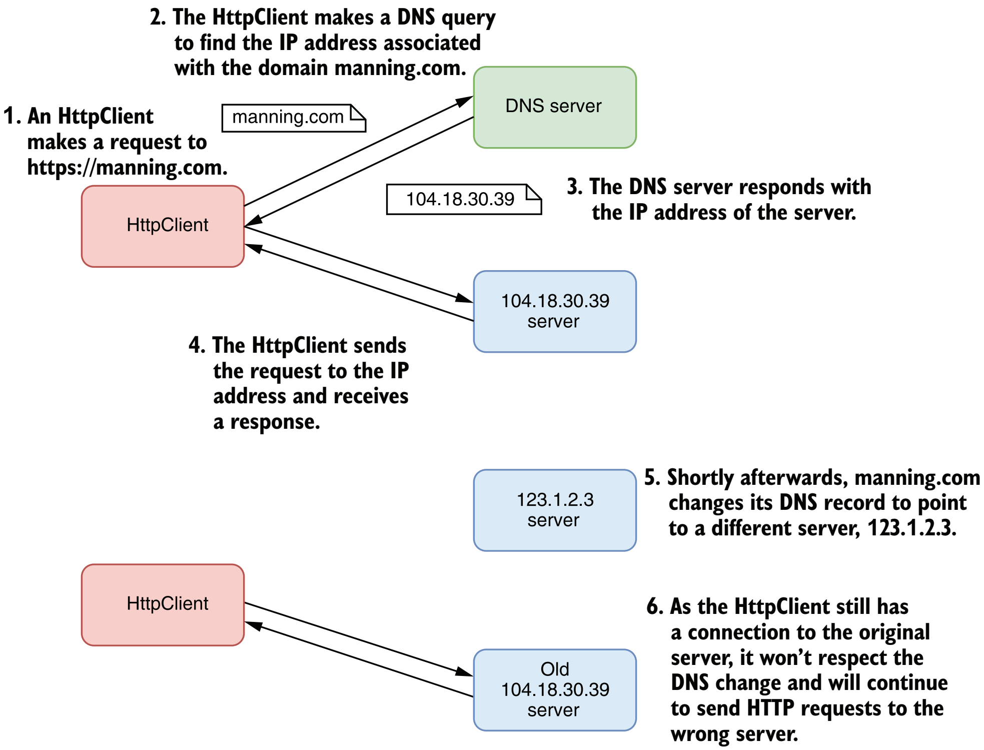

2. The HttpClient makes a DNS query to find the IP address associated with the domain manning.com.
1. An HttpClient makes a request to https://manning.com.
manning.com
DNS server
HttpClient
104.18.30.39
3. The DNS server responds with the IP address of the server.
104.18.30.39 server
4. The HttpClient sends the request to the IP address and receives a response.
123.1.2.3 server
5. Shortly afterwards, manning.com changes its DNS record to point to a different server, 123.1.2.3.
HttpClient
6. As the HttpClient still has a connection to the original server, it won't respect the DNS change and will continue to send HTTP requests to the wrong server.
Old 104.18.30.39 server

Figure 21.3 `HttpClient` **does a DNS lookup before establishing a connection, to determine the IP address associated with a hostname. If the DNS record for a hostname changes, a singleton** `HttpClient` **will not detect it and will continue sending requests to the original server it connected to.**

> **NOTE** `HttpClient` won't respect a DNS change while the original connection exists. If the original connection is closed (for example, if the original server goes offline), it will respect the DNS change as it must establish a new connection.

[1] Azure Traffic Manager, for example, uses DNS to route requests. You can read more about how it works at https://azure.microsoft.com/en-gb/services/traffic-manager/.

It seems like you're damned if you, and you're damned if you don't! Luckily, IHttpClientFactory can take care of all this for you.

21.2 Creating HttpClients with IHttpClientFactory

In this section you'll learn how you can use IHttpClientFactory to avoid the common pitfalls of HttpClient. I'll show several patterns you can use to create HttpClients:

- Using CreateClient() as a drop-in replacement for HttpClient
- Using *named clients* to centralize the configuration of an HttpClient used to call a specific third-party API
- Using *typed clients* to encapsulate the interaction with a third-party API for easier consumption by your code

IHttpClientFactory was introduced in .NET Core 2.1. It makes it easier to create HttpClient instances *correctly*, instead of relying on either of the faulty approaches I discussed in section 21.1. It also makes it easier to configure multiple HttpClients and allows you to create a middleware pipeline for outgoing requests.

Before we look at how IHttpClientFactory achieves all that, we will look a little closer at how HttpClient works under the hood.

21.2.1 Using IHttpClientFactory to manage HttpClientHandler lifetime

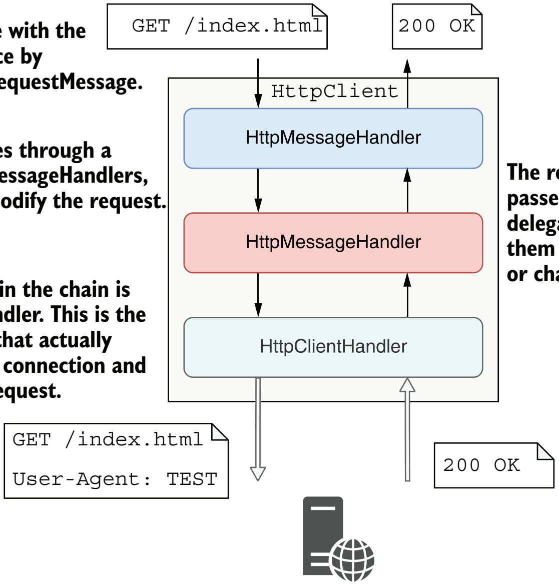
In this section we'll look at the handler pipeline used by HttpClient. You'll see how IHttpClientFactory manages the lifetime of this pipeline and how this enables the factory to avoid both socket exhaustion and DNS issues.

The HttpClient class you typically use to make HTTP requests is responsible for orchestrating requests, but it isn't responsible for making the raw connection itself. Instead, the HttpClient calls into a pipeline of HttpMessageHandler, at the end of which is an HttpClientHandler, which makes the actual connection and sends the HTTP request, as shown in figure 21.4.

This configuration is very reminiscent of the middleware pipeline used by ASP.NET Core applications, but this is an *outbound* pipeline. When an HttpClient makes a request, each handler gets a chance to modify the request before the final HttpClientHandler makes the real HTTP request. Each handler, in turn, then gets a chance to view the response after it's received.

TIP You'll see an example of using this handler pipeline for cross-cutting concerns in section 21.2.4 when we add a transient error handler.

The issues of socket exhaustion and DNS I described in section 21.1 are both related to the disposal of the HttpClientHandler at the end of the handler pipeline. By default, when you dispose of an HttpClient, you dispose of the handler pipeline too. IHttpClientFactory separates the lifetime of the HttpClient from the underlying HttpClientHandler.



A request is made with the HttpClient instance by sending an HttpRequestMessage.

The request passes through a pipeline of HttpMessageHandlers, which can each modify the request.

The final handler in the chain is the HttpClientHandler. This is the primary handler that actually makes the TCP/IP connection and sends the HTTP request.

The response message passes back through each delegating handler, giving them a chance to inspect or change the response.

Figure 21.4 Each `HttpClient` contains a pipeline of `HttpMessageHandlers`. The final handler is an `HttpClientHandler`, which makes the connection to the remote server and sends the HTTP request. This configuration is similar to the ASP.NET Core middleware pipeline, and it allows you to make cross-cutting adjustments to outgoing requests.

Separating the lifetime of these two components enables the `IHttpClientFactory` to solve the problems of socket exhaustion and DNS rotation. It achieves this in two ways:

- *By creating a pool of available handlers*—Socket exhaustion occurs when you dispose of an `HttpClientHandler`, due to the `TIME_WAIT` problem described previously. `IHttpClientFactory` solves this by creating a *pool* of handlers.

  `IHttpClientFactory` maintains an *active* handler that is used to create all `HttpClients` for two minutes. When the `HttpClient` is disposed of, the underlying handler *isn't* disposed of, so the connection isn't closed. As a result, socket exhaustion isn't a problem.
- *By periodically disposing of handlers*—Sharing handler pipelines solves the socket exhaustion problem, but it doesn't solve the DNS issue. To work around this, the `IHttpClientFactory` periodically (every two minutes) creates a new active `HttpClientHandler` that is used for each `HttpClient` created subsequently. As these `HttpClients` are using a new handler, they make a new TCP/IP connection, so DNS changes are respected.

  `IHttpClientFactory` disposes of "expired" handlers periodically in the background once they are no longer used by an `HttpClient`. This ensures there are only ever a limited number of connections in use by your application's HttpClients.[2]

Rotating handlers with IHttpClientFactory solves both of the issues we've discussed. Another bonus is that it's easy to replace existing uses of HttpClient with IHttpClientFactory.

IHttpClientFactory is included by default in ASP.NET Core; you just need to add it to your application's services in the ConfigureServices() method of Startup.cs:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient()
}
```

This registers the IHttpClientFactory as a singleton in your application, so you can inject it into any other service. For example, the following listing shows how you can replace the HttpClient approach from listing 21.2 with a version that uses IHttpClientFactory.

Listing 21.3 Using IHttpClientFactory to create an HttpClient

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHttpClientFactory _factory;
    public ValuesController(IHttpClientFactory factory)
    {
        _factory = factory;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        HttpClient client = _factory.CreateClient();

        client.BaseAddress =
            new Uri("https://api.exchangeratesapi.io");
        client.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "ExchangeRateViewer");

        var response = await client.GetAsync("latest");

        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

Inject the IHttpClientFactory using DI.

Create an HttpClient instance with an HttpClientHandler managed by the factory.

Configure the HttpClient for calling the API as before.

Use the HttpClient in exactly the same way as you would otherwise.

[2] I wrote a blog post that looks in depth at how IHttpClientFactory achieves this rotation. This is a detailed post, but it may be of interest to those who like to know how things are implemented behind the scenes. See "Exploring the code behind IHttpClientFactory in depth" at http://mng.bz/8NRK.

The immediate benefit of using `IHttpClientFactory` in this way is efficient socket and DNS handling. Minimal changes should be required to take advantage of this pattern, as the bulk of your code stays the same. This makes it a good option if you're refactoring an existing app.

`SocketsHttpHandler` vs. `IHttpClientFactory`

The limitations of `HttpClient` described in section 21.1 apply specifically to the `HttpClientHandler` at the end of the `HttpClient` handler pipeline. `IHttpClientFactory` provides a mechanism for managing the lifetime and reuse of `HttpClientHandler` instances.

In .NET Core 2.1, a replacement for the `HttpClientHandler` was introduced: `SocketsHttpHandler`. This handler has several advantages, most notably performance benefits and consistency across platforms. The `SocketsHttpHandler` can *also* be configured to use connection pooling and recycling, just like `IHttpClientFactory`.

So if `HttpClient` can already use connection pooling, is it worth using `IHttpClientFactory`? In most cases, I would say yes. You must manually configure connection pooling with `SocketsHttpHandler`, and `IHttpClientFactory` has additional features such as named clients and typed clients.

Nevertheless, if you're working in a non-DI scenario, where you can't use `IHttpClientFactory`, be sure to enable the `SocketsHttpHandler` connection pooling as described in this post by Steve Gordon, titled "HttpClient connection pooling in .NET Core": http://mng.bz/E27q.

Managing the socket issue is one big advantage of using `IHttpClientFactory` over `HttpClient`, but it's not the only benefit. You can also use `IHttpClientFactory` to clean up the client configuration, as you'll see in the next section.

21.2.2 Configuring named clients at registration time

In this section you'll learn how to use the Named Client pattern with `IHttpClientFactory`. This pattern encapsulates the logic for calling a third-party API in a single location, making it easier to use the `HttpClient` in your consuming code.

Using `IHttpClientFactory` solves the technical issues I described in section 21.1, but the code in listing 21.3 is still pretty messy in my eyes. That's primarily because you must configure the `HttpClient` to point to your service before you use it. If you need to create an `HttpClient` to call the API in more than one place in your application, you must *configure* it in more than one place too.

`IHttpClientFactory` provides a convenient solution to this problem by allowing you to centrally configure *named clients*. These clients have a `string` name and a configuration function that runs whenever an instance of the named client is requested. You can define multiple configuration functions that run in sequence to configure your new `HttpClient`.

For example, the following listing shows how to register a named client called `"rates"`. This client is configured with the correct `BaseAddress` and sets default headers that are to be sent with each outbound request.

Listing 21.4 Configuring a named client using `IHttpClientFactory` in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("rates", (HttpClient client) =>
    {
        client.BaseAddress =
            new Uri("https://api.exchangeratesapi.io");
        client.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "ExchangeRateViewer");
    })
    .ConfigureHttpClient((HttpClient client) => {})
    .ConfigureHttpClient(
        (IServiceProvider provider, HttpClient client) => {});
}
```

Provide a name for the client and a configuration function.

The configuration function runs every time the named HttpClient is requested.

You can add additional configuration functions for the named client, which run in sequence.

Additional overloads exist that allow access to the DI container when creating a named client.

Once you have configured this named client, you can create it from an `IHttpClientFactory` instance using the name of the client, `"rates"`. The following listing shows how you could update listing 21.3 to use the named client configured in listing 21.4.

Listing 21.5 Using `IHttpClientFactory` to create a named `HttpClient`

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHttpClientFactory _factory;
    public ValuesController(IHttpClientFactory factory)
    {
        _factory = factory;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        HttpClient client = _factory.CreateClient("rates");

        var response = await client.GetAsync("latest");

        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

Inject the IHttpClientFactory using DI.

Request the named client called "rates" and configure it as defined in ConfigureServices().

Use the HttpClient in the same way as before.

NOTE You can still create unconfigured clients using `CreateClient()` without a name. Be aware that if you pass an unconfigured name, such as `CreateClient("MyRates")`, the client returned will be unconfigured. Take care—client names are case-sensitive, so `"rates"` is a different client than `"Rates"`.

Named clients allow you to centralize your `HttpClient` configuration in one place, removing the responsibility of *configuring* the client from your consuming code. But you're still working with raw HTTP calls at this point—for example, providing the relative URL to call (`"/latest"`) and parsing the response. `IHttpClientFactory` includes a feature that makes it easier to clean up this code.

21.2.3 Using typed clients to encapsulate HTTP calls

A common pattern when you need to interact with an API is to encapsulate the mechanics of that interaction into a separate service. You could easily do this with the `IHttpClientFactory` features you've already seen by extracting the body of the `GetRates()` function from listing 21.5 into a separate service. But `IHttpClientFactory` has deeper support for this pattern too.

`IHttpClientFactory` supports *typed clients*. A typed client is a class that accepts a configured `HttpClient` in its constructor. It uses the `HttpClient` to interact with the remote API and exposes a clean interface for consumers to call. All of the logic for interacting with the remote API is encapsulated in the typed client, such as which URL paths to call, which HTTP verbs to use, and the types of responses the API returns. This encapsulation makes it easier to call the third-party API from multiple places in your app by using the typed client.

For example, the following listing shows an example typed client for the exchange rates API shown in previous listings. It accepts an `HttpClient` in its constructor and exposes a `GetLatestRates()` method that encapsulates the logic for interacting with the third-party API.

Listing 21.6 Creating a typed client for the exchange rates API

```
public class ExchangeRatesClient
{
    private readonly HttpClient _client;
    public ExchangeRatesClient(HttpClient client)
    {
        _client = client;
    }

    public async Task<string> GetLatestRates()
    {
        var response = await _client.GetAsync("latest");
        response.EnsureSuccessStatusCode();

        return await response.Content.ReadAsString();
    }
}
```

Inject an HttpClient using DI instead of an IHttpClientFactory.

The GetLatestRates() logic encapsulates the logic for interacting with the API.

Use the HttpClient the same way as before.

We can then inject this ExchangeRatesClient into consuming services, and they don't need to know anything about how to make HTTP requests to the remote service; they just need to interact with the typed client. We can update listing 21.3 to use the typed client as shown in the following listing, at which point the GetRates() action method becomes trivial.

Listing 21.7 Consuming a typed client to encapsulate calls to a remote HTTP server

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly ExchangeRatesClient _ratesClient;
    public ValuesController(ExchangeRatesClient ratesClient)
    {
        _ratesClient = ratesClient;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        return await _ratesClient.GetLatestRates();
    }
}
```

Inject the typed client in the constructor.

Call the typed client's API. The typed client handles making the correct HTTP requests.

You may be a little confused at this point: I haven't mentioned how IHttpClientFactory is involved yet!

The ExchangeRatesClient takes an HttpClient in its constructor. IHttpClientFactory is responsible for creating the HttpClient, configuring it to call the remote service and injecting it into a new instance of the typed client.

You can register the ExchangeRatesClient as a typed client and configure the HttpClient that is injected in ConfigureServices, as shown in the following listing. This is very similar to configuring a named client, so you can register additional configuration for the HttpClient that will be injected into the typed client.

Listing 21.8 Registering a typed client with HttpClientFactory in Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient<ExchangeRatesClient>
        (HttpClient client) =>
    {
        client.BaseAddress =
            new Uri("https://api.exchangeratesapi.io");
        client.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "ExchangeRateViewer");
    })
    .ConfigureHttpClient((HttpClient client) => {});
}
```

Register a typed client using the generic AddHttpClient method.

You can provide an additional configuration function for the HttpClient that will be injected.

As for named clients, you can provide multiple configuration methods.

TIP You can think of a typed client as a wrapper around a named client. I'm a big fan of this approach as it encapsulates all the logic for interacting with a remote service in one place. It also avoids the "magic strings" that you use with named clients, removing the possibility of typos.

Another option when registering typed clients is to register an interface in addition to the implementation. This is often a good practice, as it makes it much easier to test consuming code. For example, if the typed client in listing 21.6 implemented the interface IExchangeRatesClient, you could register the interface and typed client implementation using

```
services.AddHttpClient<IExchangeRatesClient, ExchangeRatesClient>()
```

You could then inject this into consuming code using the interface type:

```
public ValuesController(IExchangeRatesClient ratesClient)
```

Another commonly used pattern is to not provide any configuration for the typed client in ConfigureServices(). Instead, you could place that logic in the constructor of your ExchangeRatesClient using the injected HttpClient:

```
public class ExchangeRatesClient
{
    private readonly HttpClient _client;
    public ExchangeRatesClient(HttpClient client)
    {
        _client = client;
        _client.BaseAddress = new Uri("https://api.exchangeratesapi.io");
    }
}
```

This is functionally equivalent to the approach shown in listing 21.8. It's a matter of taste where you'd rather put the configuration for your HttpClient. If you take this approach, you don't need to provide a configuration lambda in ConfigureServices:

```
services.AddHttpClient<ExchangeRatesClient>();
```

Named clients and typed clients are convenient for managing and encapsulating HttpClient configuration, but IHttpClientFactory brings another advantage we haven't looked at yet: it's easier to extend the HttpClient handler pipeline.

21.3 Handling transient HTTP errors with Polly

In this section you'll learn how to handle a very common scenario: "transient" errors when you make calls to a remote service, caused by an error in the remote server or temporary network issues. You'll see how to use IHttpClientFactory to handle cross-cutting concerns like this by adding handlers to the HttpClient handler pipeline.

In section 21.2.1 I described HttpClient as consisting of a "pipeline" of handlers. The big advantage of this pipeline, much like the middleware pipeline of your application, is that it allows you to add cross-cutting concerns to all requests. For example, IHttpClientFactory automatically adds a handler to each HttpClient that logs the status code and duration of each outgoing request.

As well as logging, another very common requirement is to handle transient errors when calling an external API. Transient errors can happen when the network drops out, or if a remote API goes offline temporarily. For transient errors, simply trying the request again can often succeed, but having to *manually* write the code to do so is cumbersome.

ASP.NET Core includes a library called Microsoft.Extensions.Http.Polly that makes handling transient errors easier. It uses the popular open source library Polly (https://github.com/App-vNext/Polly) to automatically retry requests that fail due to transient network errors.

Polly is a mature library for handling transient errors that includes a variety of different error-handling strategies, such as simple retries, exponential backoff, circuit breaking, bulkhead isolation, and many more. Each strategy is explained in detail at https://github.com/App-vNext/Polly, so be sure to read about the benefits and trade-offs when selecting a strategy.

To provide a taste of what's available, we'll add a simple retry policy to the ExchangeRatesClient shown in section 21.2. If a request fails due to a network problem, such as a timeout or a server error, we'll configure Polly to automatically retry the request as part of the handler pipeline, as shown in figure 21.5.

To add transient error handling to a named client or HttpClient, follow these steps:

1. Install the Microsoft.Extensions.Http.Polly NuGet package in your project by running dotnet add package Microsoft.Extensions.Http.Polly, or by using the NuGet explorer in Visual Studio, or by adding a <PackageReference> element to your project file as follows:

   ```
   <PackageReference Include="Microsoft.Extensions.Http.Polly"
       Version="5.0.0" />
   ```

2. Configure a named or typed client as shown in listings 21.5 and 21.7.
3. Configure a transient error-handling policy for your client as shown in listing 21.9.

Listing 21.9 Configuring a transient error-handling policy for a typed client in Startup.cs

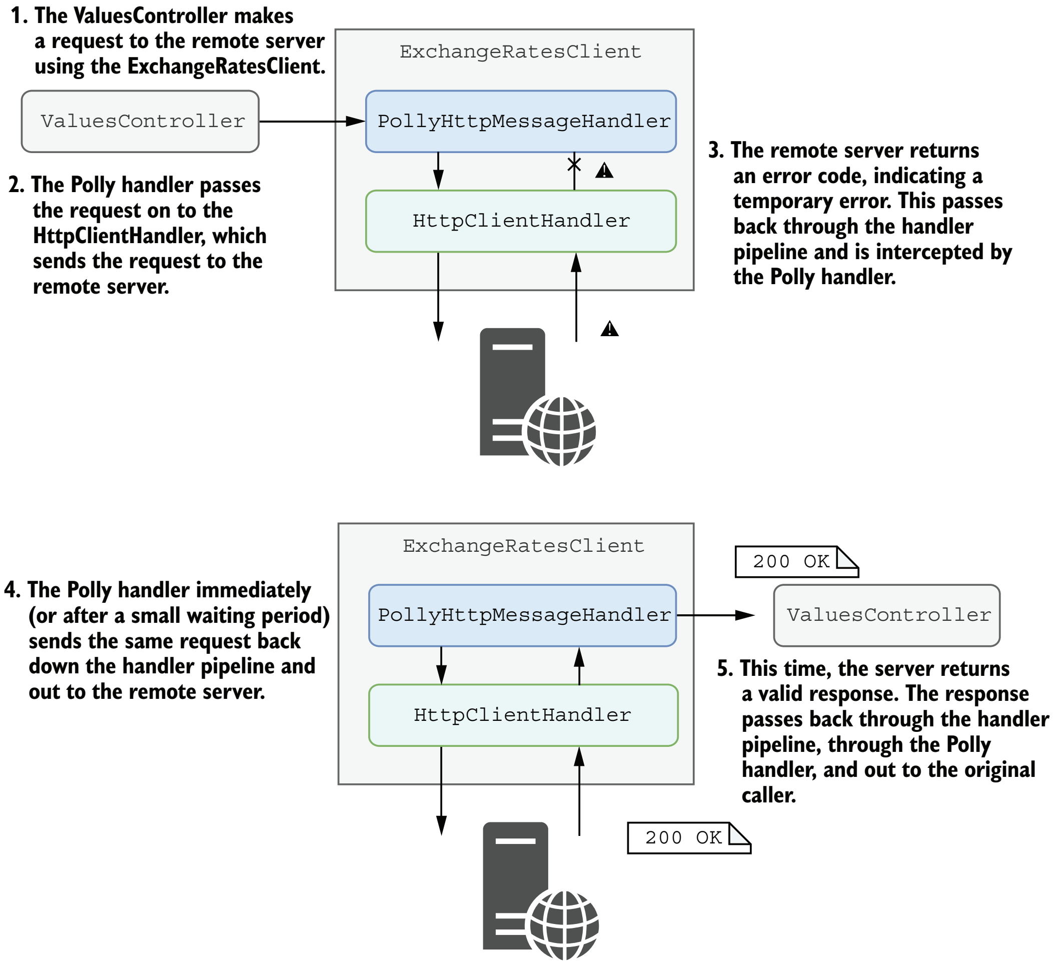
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient<ExchangeRatesClient>()
        .AddTransientHttpErrorPolicy(policy =>
            policy.WaitAndRetryAsync(new[] {
                TimeSpan.FromMilliseconds(200),
                TimeSpan.FromMilliseconds(500),
                TimeSpan.FromSeconds(1)
            })
        );
}
```

You can add transient error handlers to named or typed clients.

Use the extension methods provided by the NuGet package to add transient error handlers.

Configures a policy that waits and retries three times if an error occurs

Configure the retry policy used by the handler. There are many types of policies to choose from.



1. The ValuesController makes a request to the remote server using the ExchangeRatesClient.

2. The Polly handler passes the request on to the HttpClientHandler, which sends the request to the remote server.

3. The remote server returns an error code, indicating a temporary error. This passes back through the handler pipeline and is intercepted by the Polly handler.

4. The Polly handler immediately (or after a small waiting period) sends the same request back down the handler pipeline and out to the remote server.

5. This time, the server returns a valid response. The response passes back through the handler pipeline, through the Polly handler, and out to the original caller.

Figure 21.5 Using the `PolicyHttpMessageHandler` to handle transient errors. If an error occurs when calling the remote API, the Polly handler will automatically retry the request. If the request then succeeds, the result is passed back to the caller. The caller didn't have to handle the error themselves, making it simpler to use the `HttpClient` while remaining resilient to transient errors.

In the preceding listing we configure the error handler to catch transient errors and retry three times, waiting an increasing amount of time between requests. If the request fails on the third try, the handler will ignore the error and pass it back to the client, just as if there was no error handler at all. By default, the handler will retry any request that either

- Throws an `HttpRequestException`, indicating an error at the protocol level, such as a closed connection
- Returns an HTTP 5xx status code, indicating a server error at the API
- Returns an HTTP 408 status code, indicating a timeout

TIP If you want to handle more cases automatically, or to restrict the responses that will be automatically retried, you can customize the selection logic as described in the "Polly and HttpClientFactory" documentation on GitHub: http://mng.bz/NY7E.

Using standard handlers like the transient error handler allows you to apply the same logic across all requests made by a given `HttpClient`. The exact strategy you choose will depend on the characteristics of both the service and the request, but a good retry strategy is a must whenever you interact with potentially unreliable HTTP APIs.

The Polly error handler is an example of an optional `HttpMessageHandler` that you can plug in to your `HttpClient`, but you can also create your own custom handler. In the next section you'll see how to create a handler that adds a header to all outgoing requests.

21.4 Creating a custom HttpMessageHandler

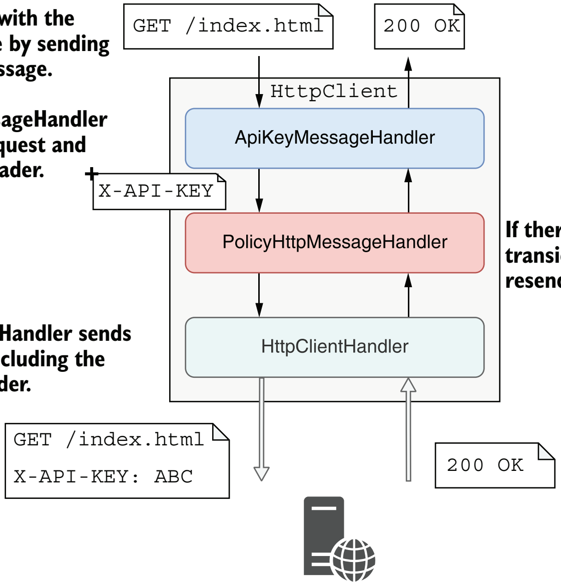
Most third-party APIs will require some form of authentication when you're calling them. For example, many services require you attach an API key to an outgoing request, so that the request can be tied to your account. Instead of having to remember to manually add this header for every request to the API, you could configure a custom `HttpMessageHandler` to automatically attach the header for you.

NOTE More complex APIs may use JSON Web Tokens (JWT) obtained from an identity provider. If that's the case, consider using the open source IdentityModel library (https://identitymodel.readthedocs.io), which provides integration points for ASP.NET Core Identity and `HttpClientFactory`.

You can configure a named or typed client using `IHttpClientFactory` to use your API-key handler as part of the `HttpClient`'s handler pipeline, as shown in figure 21.6. When you use the `HttpClient` to send a message, the `HttpRequestMesssage` is passed through each handler in turn. The API-key handler adds the extra header and passes the request to the next handler in the pipeline. Eventually the `HttpClientHandler` makes the network request to send the HTTP request. After the response is received, each handler gets a chance to inspect (and potentially modify) the response.

To create a custom `HttpMessageHandler` and add it to a typed or named client's pipeline, you can follow these steps:

1. Create a custom handler by deriving from the `DelegatingHandler` base class.
2. Override the `SendAsync()` method to provide your custom behavior. Call `base.SendAsync()` to execute the remainder of the handler pipeline.
3. Register your handler with the DI container. If your handler does not require state, you can register it as a singleton service; otherwise you should register it as a transient service.
4. Add the handler to one or more of your named or typed clients by calling `AddHttpMessageHandler<T>()` on an `IHttpClientBuilder`, where `T` is your



A request is made with the HttpClient instance by sending an HttpRequestMessage.

The ApiKeyMessageHandler modifies the request and adds the API header.

If there is an error, the transient error handler resends the request.

The HttpClientHandler sends the request, including the additional header.

Figure 21.6 You can use a custom `HttpMessageHandler` to modify requests before they're sent to third-party APIs. Every request passes through the custom handler before the final handler (the `HttpClientHandler`) sends the request to the HTTP API. After the response is received, each handler gets a chance to inspect and modify the response.

handler type. The order in which you register handlers dictates the order in which they will be added to the `HttpClient` handler pipeline. You can add the same handler type more than once in a pipeline if you wish, and to multiple typed or named clients.

The following listing shows an example of a custom `HttpMessageHandler` that adds a header to every outgoing request. We'll use the custom `"X-API-KEY"` header in this example, but the header you need will vary depending on the third-party API you're calling. This example uses strongly typed configuration to inject the secret API key, as you saw in chapter 10.

Listing 21.10 Creating a custom `HttpMessageHandler`

```
public class ApiKeyMessageHandler : DelegatingHandler
{
    private readonly ExchangeRateApiSettings _settings;
    public ApiKeyMessageHandler(
        IOptions<ExchangeRateApiSettings> settings)
    {
        _settings = settings.Value;
    }
```

Custom HttpMessageHandlers should derive from DelegatingHandler.

Inject the strongly typed configuration values using dependency injection.

```
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        request.Headers.Add("X-API-KEY", _settings.ApiKey);

        HttpResponseMessage response =
            await base.SendAsync(request, cancellationToken);

        return response;
    }
}
```

Override the SendAsync method to implement the custom behavior.

Add the extra header to all outgoing requests.

Call the remainder of the pipeline and receive the response.

You could inspect or modify the response before returning it.

To use the handler, you must register it with the DI container and add it to a named or typed client. In the following listing, we add it to the `ExchangeRatesClient`, along with the transient error handler we registered in listing 21.8. This creates a pipeline similar to that shown in figure 21.6.

Listing 21.11 Registering a custom handler in `Startup.ConfigureServices`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<ApiKeyMessageHandler>();

    services.AddHttpClient<ExchangeRatesClient>()
        .AddHttpMessageHandler<ApiKeyMessageHandler>()
        .AddTransientHttpErrorPolicy(policy =>
            policy.WaitAndRetryAsync(new[] {
                TimeSpan.FromMilliseconds(200),
                TimeSpan.FromMilliseconds(500),
                TimeSpan.FromSeconds(1)
            })
        );
}
```

Register the custom handler with the DI container.

Configure the typed client to use the custom handler.

Add the transient error handler. The order in which they are registered dictates their order in the pipeline.

Whenever you make a request using the typed client `ExchangeRatesClient`, you can be sure the API key will be added and that transient errors will be handled automatically for you.

That brings us to the end of this chapter on `IHttpClientFactory`. Given the difficulties in using `HttpClient` correctly that I showed in section 21.1, you should always favor `IHttpClientFactory` where possible. As a bonus, `IHttpClientFactory` allows you to easily centralize your API configuration using named clients and to encapsulate your API interactions using typed clients.

Summary

- Use the `HttpClient` class for calling HTTP APIs. You can use it to make HTTP calls to APIs, providing all the headers and body to send in a request, and reading the response headers and data you get back.

- HttpClient uses a pipeline of handlers, consisting of multiple HttpMessage-Handlers, connected in a similar way to the middleware pipeline used in ASP.NET Core. The final handler is the HttpClientHandler, which is responsible for making the network connection and sending the request.
- HttpClient implements IDisposable, but you shouldn't typically dispose of it. When the HttpClientHandler that makes the TCP/IP connection is disposed of, it keeps a connection open for the TIME_WAIT period. Disposing of many HttpClients in a short period of time can lead to socket exhaustion, preventing a machine from handling any more requests.
- Prior to .NET Core 2.1, the advice was to use a single HttpClient for the lifetime of your application. Unfortunately, a singleton HttpClient will not respect DNS changes, which are commonly used for traffic management in cloud environments.
- IHttpClientFactory solves both these problems by managing the lifetime of the HttpMessageHandler pipeline. You can create a new HttpClient by calling CreateClient(), and IHttpClientFactory takes care of disposing of the handler pipeline when it is no longer in use.
- You can centralize the configuration of an HttpClient in ConfigureServices() using *named* clients by calling AddHttpClient("test", c => {}). You can then retrieve a configured instance of the client in your services by calling IHttp-ClientFactory.CreateClient("test").
- You can create a *typed* client by injecting an HttpClient into a service, T, and configuring the client using AddHttpClient<T>(c => {}). Typed clients are great for abstracting the HTTP mechanics away from consumers of your client.
- You can use the Microsoft.Extensions.Http.Polly library to add transient HTTP error handling to your HttpClients. Call AddTransientHttpErrorPolicy() when configuring your IHttpClientFactory in ConfigureServices, and provide a Polly policy to control when errors should be automatically handled and retried.
- It's common to use a simple retry policy to try making a request multiple times before giving up and returning an error. When designing a policy, be sure to consider the impact of your policy; in some circumstances it may be better to fail quickly instead of retrying a request that is never going to succeed. Polly includes additional policies such as circuit-breakers to create more advanced approaches.
- By default, the transient error-handling middleware will handle connection errors, server errors that return a 5xx error code, and 408 (timeout) errors. You can customize this if you want to handle additional error types, but ensure that you only retry requests that are safe to do so.
- You can create a custom HttpMessageHandler to modify each request made through a named or typed client. Custom handlers are good for implementing cross-cutting concerns such as logging, metrics, and authentication.

- To create a custom HttpMessageHandler, derive from DelegatingHandler and override the SendAsync() method. Call base.SendAsync() to send the request to the next handler in the pipeline and finally to the HttpClientHandler, which makes the HTTP request.
- Register your custom handler in the DI container as either a transient or a singleton. Add it to a named or typed client using AddHttpMessageHandler<T>(). The order in which you register the handler in the IHttpClientBuilder is the order in which the handler will appear in the HttpClient handler pipeline.